EECS 470 Lab 5 Assignment

Note:

- The lab should be completed individually
- The lab check off is due by **Friday, October 10th**

1 Introduction

In this lab assignment you will be designing a generically-sized First-In First-Out circular buffer (a `FIFO` buffer). A hardware `FIFO` buffer is just like a software FIFO queue: the write operation does a push to the back (tail) of the queue, and a read returns and pops the front (head) of the queue. Circular buffers are a fundamental hardware component and used throughout industry and in your final project. You will be writing a parameterized version here that you can reference and extend for the final project.

2 FIFO Description

Our `FIFO` module has the following interface:

```
module FIFO #(
    parameter DEPTH = 16,
    parameter WIDTH = 32,
    parameter MAX_CNT = 3,
    localparam CNT_BITS = $clog2(MAX_CNT+1)
) (
    input                          clock,
    input                          reset,
    input                          wr_en,
    input                          rd_en,
    input            [WIDTH-1:0] wr_data,
    output logic                   wr_valid,
    output logic                   rd_valid,
    output logic     [WIDTH-1:0] rd_data,
    output logic [CNT_BITS-1:0] spots,
    output logic                   full
);
```

A FIFO uses a circular buffer internally with two pointers to its head and tail. These are pointers, so if your buffer can hold 32 entries, the pointers must be at least $log_2 32 = 5$ bits each (be careful with rounding if this is a non-power of two). The *head* pointer tracks the head of the queue, where you can read and pop from, and the *tail* pointer tracks the tail of the queue, where you write new data.

We use read and write enable bits on the input to say whether someone is reading or writing, but they don't know if this was successful until they see the matching valid bit in the output.

We need the valid bits since we can't *read* when the queue is empty and we can't *write* when the queue is full.

But we also need to keep track of whether we're empty or full.

A FIFO should start in an "empty" state with both `head` and `tail` at 0. So a FIFO is empty when `head == tail`, but it's also *full* when `head == tail`! This is a problem, and there are a few good ways to solve it:

1. (easiest) We can keep track of the number of entries in the queue with a simple counter. Then we're empty when `entries == 0` and full when `entries == SIZE`.

2. (easy) Make the FIFO one larger than your desired size, and use `head == tail` for empty and `head == tail + 1` for full (but be careful with overflow issues!).
3. (medium) We can use valid bits to say whether each entry in the queue is valid or not. Then we're empty when `ALL(valid_bits) == 0` and full when `ALL(valid_bits) == 1`
4. (medium) Add one extra bit to your head and tail pointers so you can check the MSB to know if it is full or empty. You don't use this extra bit to index into the FIFO, but you check it when the lower bits are equal to know if it is full or empty. The MSBs in the head and tail being equal mean empty, and different is full.
5. (harder) We can keep using `head == tail`, but add extra logic to our full and empty indicator variables to track whether our previous state increased or decreased the number of entries.
6. (harder) Similar to the previous option, have a one-bit flip-flop tracking your last operation (read or write). Set it to 1 when you do a write without reading (growing your FIFO), 0 when you read without writing (shrinking the FIFO), and otherwise keep it the same. You are full when `head == tail` and the last operation was a write, and empty when `head == tail` and the last operation was a read.

The first is the easiest but a bit redundant and requires maintaining an extra counter. The second requires the least external tracking logic, but means you can never fully utilize the space in the FIFO. The third has high register storage/overhead, but is quicker. And the last has the lowest cost in bits, but is harder to reason about and get correct.

You may implement any of these strategies in your FIFO module for this lab, or something else if you feel so inclined.

We also add an extra complication of a "spots" output, which we use to tell other modules when we're almost full. The "spots" output indicates the number of remaining spots in the FIFO, but saturates at a max value specified by the `MAX_CNT` parameter. Saturating this counter reduces the interconnect between modules since the number of spots remaining is only relevant when it gets low. This kind of output is useful in superscalar processors, which many of you will choose for your final project.

3 Assignment

Implement the `FIFO` module with the following functionality:

1. Maintain a set of `SIZE` (default of 16) `WIDTH`-bit memory elements, which should only be updated on the positive edge of `clock`
2. When `reset` is high on the rising clock edge, the buffer should become empty (i.e. an immediately following read request would not be valid).
3. By default, `wr_valid` and `rd_valid` should be 0.
4. If `wr_en` is high at the positive edge of the clock, `FIFO` should store the value of `wr_data` to the first free index in the FIFO (you will need to maintain some sort of tail pointer for this). If there is no more space (`SIZE` values have been written and not read), `FIFO` should not modify its state and set `wr_valid` to 0; otherwise, `wr_valid` should be 1 indicating a that the write is accepted.
5. If `rd_en` is high, `FIFO` should write the oldest unread value written to it on the `rd_data` output and increment its head pointer so that the next read will output the next oldest value. If there are no values currently saved, it should set `rd_valid` to 0; otherwise, `rd_valid` should be 1 indicating that the read is accepted.
6. The output `full` should be 1 iff there is no more space in in the buffer. This indicates to external modules that they cannot write anything else to the buffer.
7. There is an additional output `spots` which indicates the number of remaining spots in the FIFO when they get below a certain threshold. This helps external modules know there is limited space remaining.

When there are more than `MAX_CNT` spots remaining, `spots` should equal `MAX_CNT`.

8. If you simultaneously read and write to an empty FIFO, then the read should be invalid because there is nothing to read. In other words, there is a minimum one cycle latency between data going in and out.
9. If you simultaneously read and write to a full FIFO, then the write should be valid because there it can write to the newly opened spot.

Your design should function correctly for any values of `SIZE`, `WIDTH`, and `MAX_CNT`. It is up to you how to implement this functionality. We recommend using a read and write pointer to keep track of where you should read from and write to next in the buffer.

Your design should pass the testbench after synthesis as well. However you don't need to worry about clock period for this lab, just make a design that works.

3.1 memDP.sv

We have included an additional module — `memDP` — which abstracts a dual-ported memory. This models a custom memory which you will often encounter in real ASIC design. We will be requiring that you use this module for all caches in your final project, and we encourage you to use it here as well. This module also forces you to think more carefully about how you are interacting with the FIFO structure, which often leads to more well thought-out designs

4 Verification

We will be using the test script to check off your design, but to help you debug your module we describe below the process for using both the makefile and JasperGold. We require that you at least open JasperGold in this lab, but you are free to use either the testbench or JasperGold to debug your design.

Both the testbench and formal verification use SystemVerilog assertions to check for proper functionality. This is provided as an example in case you wish to incorporate similar assertions in your design. The `bind` command in the testbench binds this assertion module with the DUT so we don't have to code the assertions in the FIFO module. This makes for cleaner and more reusable code.

4.1 Using the Testbench

Using the testbench involves the same Make system as all prior labs, but we have added some functionality in this lab due to the parametrization of the FIFO.

When running '`make`', set the `SIZE` by doing '`make <target> SIZE=i`' at the shell; replacing '`i`' with your desired size and '`<target>`' with whatever make command you wish to run. e.g. to run simulation with size 48, do '`make sim SIZE=48`'. Similarly, to adjust the bit-width of the FIFO entries, use '`make target WIDTH=i`' where '`i`' is the number of bits in the `rd` and `wr` lines. You can also adjust the `MAX_CNT` parameter from the command line in the same way.

The Makefile communicates these variables to the synthesis script and synthesizes separate .vg files for each fifo size to reduce compile time. See the Makefile's new `Modules with Parameters` section for details.

You might also need to add the `-B` flag (`make sim -B SIZE=i`) to tell make that it needs to recompile everything when you change the size. And the `--assume-old=` flag can avoid re-synthesizing the .vg modules when using `-B`.

4.2 Using JasperGold

JasperGold is our formal verification tool. This tool is used across industry, and experience with it is a huge plus when interviewing for jobs.

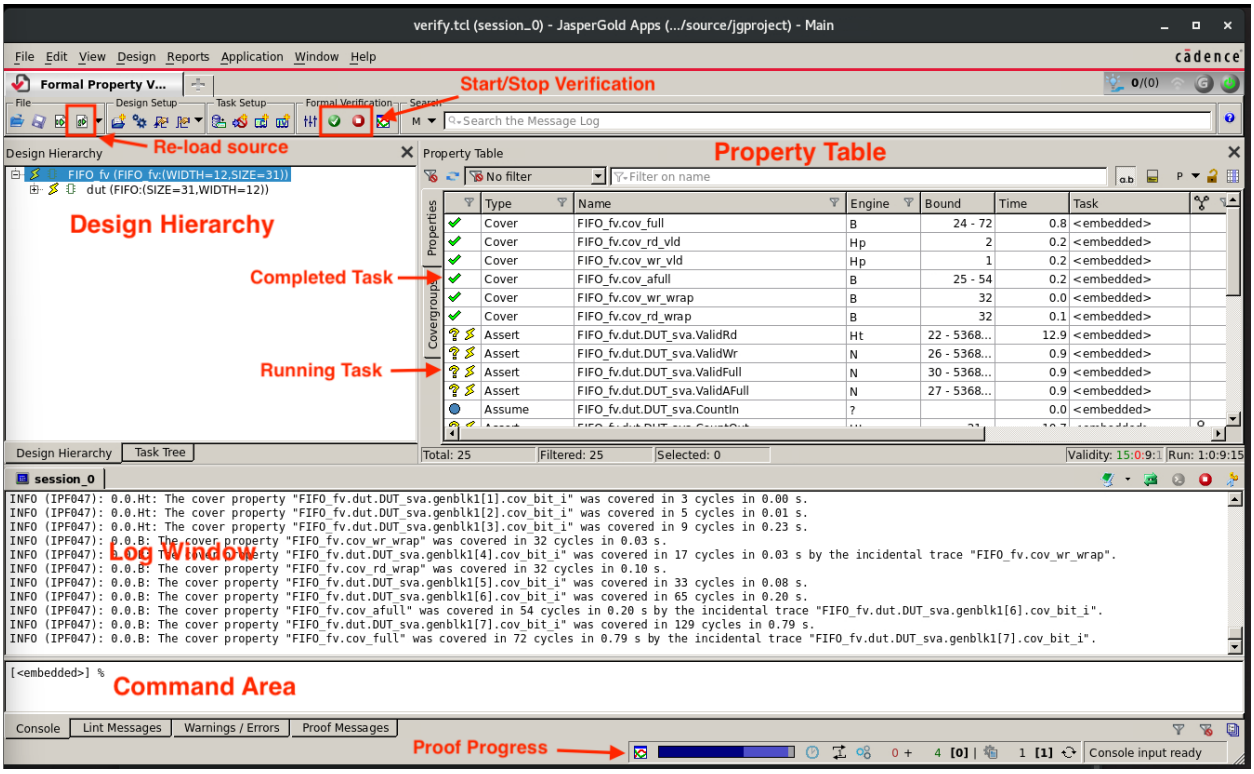

Figure 1: Main JasperGold Window

To open and run JasperGold, run the following commands from your lab 5 directory:

```
module load jasper
jaspergold -tcl verify.tcl
```

If you wish to run JasperGold without the GUI and just see the results in your terminal, add the `-batch` flag to the `jaspergold` command. This will run the tool from the command line and print its results when complete. The rest of this section assumes you are using the GUI.

After running the `jaspergold`, the JasperGold GUI will appear and begin running the proofs. If a separate window asking about credentials opens, feel free to ignore and/or close it. The JasperGold GUI has 3 main sections:

1. Design hierarchy (top left): information about the design being verified
   - Click the "+" on the left of a module to expand it
   - You can double click a module to open the source browser
2. Property table (top right): list of all properties
   - This includes cover properties, assume properties, and assertions
   - The leftmost column shows the status of each property: not started, queued, running, proven, unreachable, counterexample found (cex)
   - You can sort and filter each column, For example, you can show only failing assertions
3. Command area and logs (bottom): log messages and interface for typing commands
   - You likely won't need this much
   - But the tabs on the bottom allow you to filter different messages

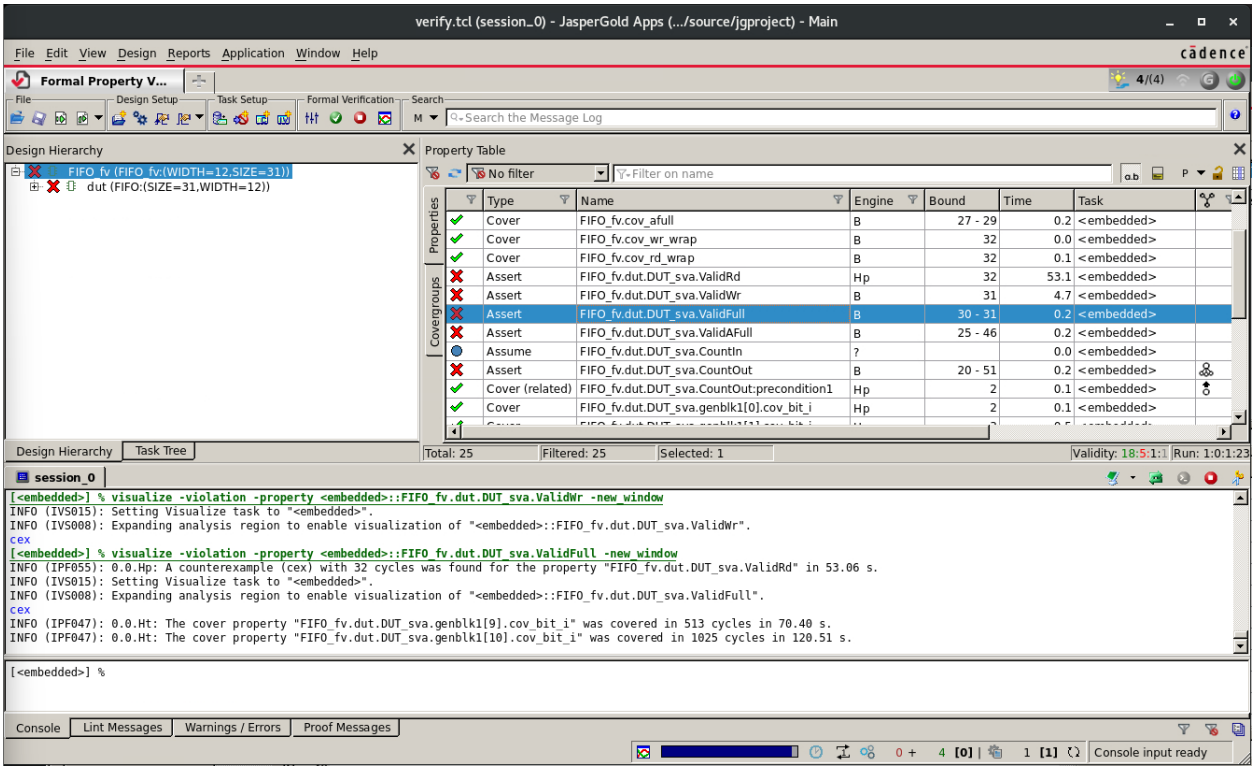

Figure 2: Example of failing assertions

Our `verify.tcl` script automatically starts the proofs. However, if you would like to stop or restart the proofs, there is a red x and green check on the toolbar above the design information / property list that can stop and start the proofs from running, respectively. If you change your design and wish to re-run the proofs without re-loading Jasper, you must first stop the current proof (by clicking th red x), and reload the source code through re-running the `verify.tcl` script by clicking the second-most left button on the bottom row of the toolbar (see picture). You can also run individual proofs by right-clicking any property and selecting "prove property".

If any assertions fail, double-click the failing property to open up a waveform of the signal trace which caused the failure. This will give you a waveform where you can double-click any signal in the wave to view its driver in the source Verilog. You can add whatever signals you wish to this waveform by dragging the signal name from the source browser and releasing it in the waveform. Double-clicking any signal in the source code takes you to its driver, allowing you to trace through the source of the misbehaving signal. You can also use the "D" and "L" signals to trace through the driver(s) and load(s), respectively, of any signal you have selected in the source code.

You can add spacers to the waveform with the "Ctrl-k" shortcut to make it more readable. Signals can also be added to the waveform through the signal browser on the right side of the waveform viewer. Select a specific module in the top half of the browser, and select the signal in the bottom half.

Note that it can be helpful to debug more than one assertion per verification run since you often have many bugs which cause different errors. This saves time through avoiding having to re-run all proofs. Just note that if you change the source code, then you cannot use the source code pane in the waveform viewer to trace through signal drivers and loads until re-running the proofs.

Once verification is complete, through either 1) proving all assertions and reaching all cover properties, or 2) manually stopping the proof, Jasper will give you a summary of how many properties passed, failed (cex), were unreachable. Proofs can take up to many hours to run depending on the size of your design and how

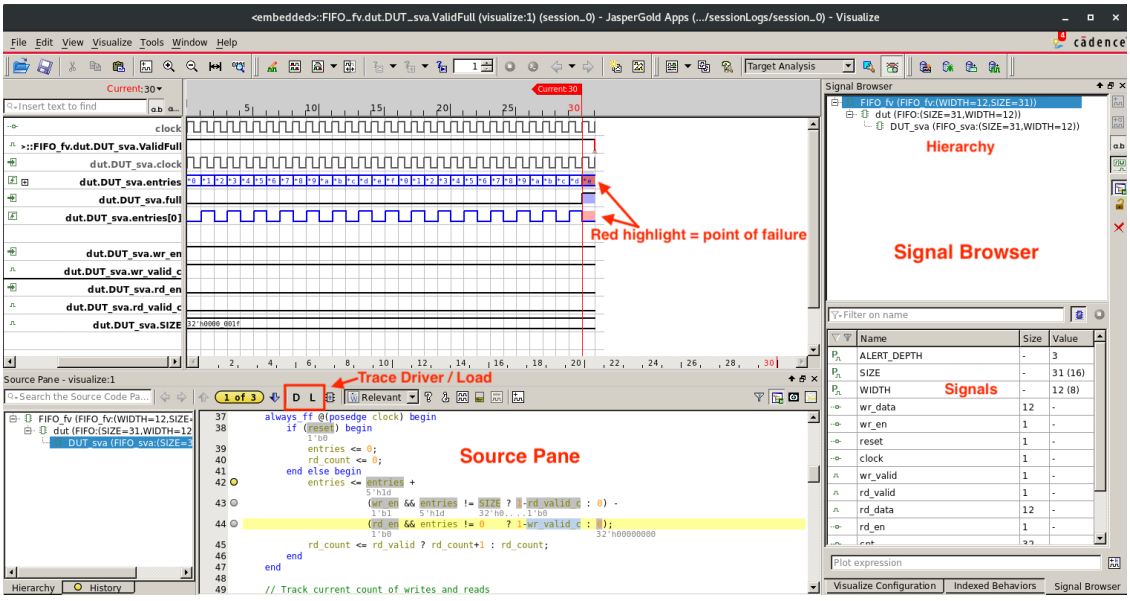

Figure 3: Annotated waveform viewer window

efficiently you code your assertions. If you cannot attain full proof, you can use your judgement to stop the proof when all assertions have attained a reasonable proof bound. This bound is the number of cycles after reset that an assertion holds or a cover property requires in order to be reached. Full proof of an assertion is when the bound is infinite. The current bound associated with each property is listed in the property table.

There are many other functions JasperGold can perform such as sequential equivalency checking, reset analysis, schematic viewing, and protocol verification. We hope that just the basic functionality described here can help you bring up and debug your modules faster through quick, visual feedback on performance:

5 Tips

- Be careful how you deal with the head and tail pointers wrapping around to the start of the buffer.
- You can use the Modulo (`%`) operator when updating the head and tail to keep them within range.
- The `$clog2()` function is helpful for finding the number of bits required to encode a number. To make sure it works with powers of 2, you should add 1 to the argument (i.e.: `clog2(NUM+1)`). The `+1` is because you need to represent the numbers `[0,NUM]` which is `NUM+1` values.
- Think about what approach you want to use to differentiate between the buffer being empty or full. Each approach comes with varying degrees of difficulty and efficiency.

6 Bonus Features

- Add the constraint that `rd_data` must be 0 when `rd_en` is 0. Can you add an assertion in `FIFO_sva.svh` to check this?
- What if you want to scan the FIFO to see if a value exists within it? This requires implementing a CAM (content addressable memory) which involves a for loop with a break statement. This will not work if you use the `memDP` module
- Define a struct (or copy one from project 3), and make the FIFO store that instead of the base logic type. Is there a way to do that without changing the code within the module at all? Note: thinking about this may be helpful for your final project

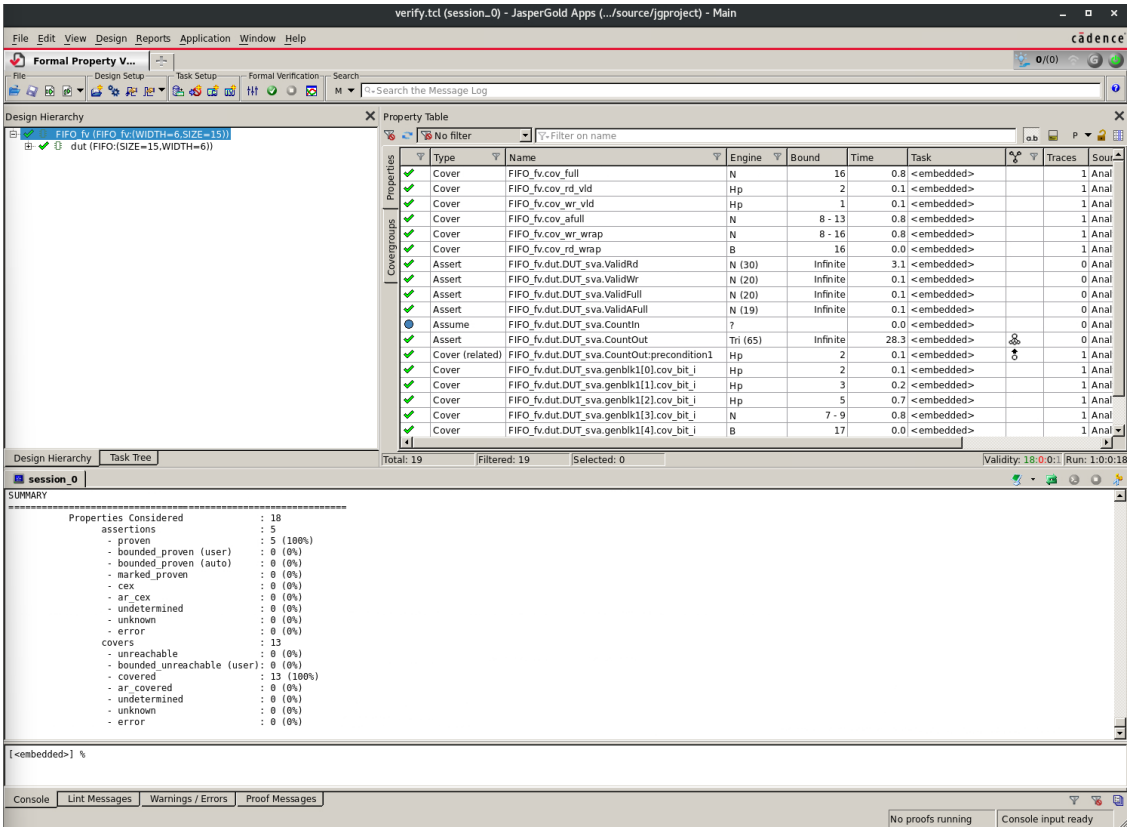

Figure 4: Complete verification with all assertions passing

7 Submission

A script has been provided (`check.sh`) that compiles and runs the testbench for a few different test `SIZE`s, `WIDTH`s, and `MAX_CNT`s. Once you are passing all of those cases and have successfully opened JasperGold, add yourself to the help queue and an instructor will check you off. Please be ready to show the script output, proof that you opened JasperGold (a screenshot or the program itself), and your `FIFO.sv` file.

Please also be ready to discuss how FIFOs are used in the final project.